

# METHODS OF MATRIX MULTIPLICATION

## AN OVERVIEW OF SEVERAL METHODS AND THEIR IMPLEMENTATION

IVO HEDTKE

VERSION 1 (JANUARY 27, 2013)

**ABSTRACT.** In this overview article we present several methods for multiplying matrices and the implementation of these methods in C. Also a little test program is given to compare their running time and the numerical stability.

The methods are: naive method, naive method working on arrays, naive method with the KAHAN trick, three methods with loop unrolling, winograd method and the scaled variant, original STRASSEN method and the STRASSEN-WINOGRAD variant.

**Please note, that this is the FIRST version. The algorithms are not well tested and the implementation is not optimized. If you like to join the project, please contact me.**

**The collection of algorithms and this document will be updated from time to time.**

### CONTENTS

1. Auxiliary Routines	2
1.1. Plus	2
1.2. Minus	2
1.3. max	2
1.4. NormInf	3
1.5. MultiplyWithScalar	3
2. Methods of Matrix Multiplication	5
2.1. The naive method	5
2.1.1. NaivStandard	5
2.1.2. NaivOnArray	5
2.1.3. NaivKahan	6
2.2. Loop unrolling	6
2.2.1. NaivLoopUnrollingTwo	6
2.2.2. NaivLoopUnrollingThree	7
2.2.3. NaivLoopUnrollingFour	9
2.3. Winograd's algorithm	10
2.3.1. WinogradOriginal	10
2.3.2. WinogradScaled	11
2.4. Strassen's algorithm	12
2.4.1. StrassenNaiv	12
2.4.2. StrassenWinograd	18
3. Tests	24
References	25

In the following text,  $A$  denotes a  $N \times P$  matrix and  $B$  denotes a  $P \times M$  matrix with entries of type double. The implementation is written in C based on the standard C99. The code and the current version of this document can be found here:

<http://www2.informatik.uni-halle.de/da/hedtke/overview-momm/>.

Feel free to use the code for anything you want. **DO NOT USE COMPILER OPTIMIZATION!**

## 1. AUXILIARY ROUTINES

1.1. **Plus.** According to the definition of the  $+$  operator for matrices

$$(A + B)_{ij} = A_{ij} + B_{ij}$$

we use a simple straightforward method to compute  $A + B$ .

---

```
void Plus(A, B, Result, N, M)
    double** A;
    double** B;
    double** Result;
    int N;
    int M;
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            Result[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

---

1.2. **Minus.** Like above we use a straightforward method to compute

$$(A - B)_{ij} = A_{ij} - B_{ij}.$$

---

```
void Minus(A, B, Result, N, M)
    double** A;
    double** B;
    double** Result;
    int N;
    int M;
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            Result[i][j] = A[i][j] - B[i][j];
        }
    }
}
```

---

1.3. **max.** As an often used auxiliary function we implement

$$\max(x, y) = \begin{cases} x & x \geq y \\ y & \text{otherwise} \end{cases}$$

for integer and double input arguments.

---

```
int max( x, y )
    int x;
    int y;
{
    if (x < y){
        return y;
    } else {
        return x;
    }
}
```

---

```

double Max( x, y )
{
    double x;
    double y;
    {
        if (x < y){
            return y;
        } else {
            return x;
        }
    }
}

```

---

1.4. **NormInf.** In the scaled variant of WINOGRADS algorithm we need the  $\infty$ -norm

$$\|A\|_{\infty} = \max_{i=1,\dots,N} \left\{ \sum_{j=1}^P |A_{ij}| \right\}$$

of a matrix  $A$ . We also use it to compare the numerical stability of the several methods in the tests.

---

```

double Abs(double a){
    if ( a < 0 ) {
        return -a;
    }
    return a;
}

double NormInf(A, N, P)
{
    double** A;
    int N;
    int P;
    {
        double Norm = 0.0;
        double aux;
        int i, j;

        for (i = 0; i < N; i++) {
            aux = 0.0;
            for (j = 0; j < P; j++) {
                aux += Abs(A[i][j]);
            }
            Norm = Max(Norm, aux);
        }

        return Norm;
    }
}

```

---

1.5. **MultiplyWithScalar.** Also in the scaled variant of WINOGRADS algorithm we need the product of a matrix  $A$  and a scalar  $\alpha$

$$(\alpha A)_{ij} = \alpha A_{ij}.$$


---

```

void MultiplyWithScalar(A, Result, N, M, alpha)
{
    double** A;
    double** Result;
    int N;
    int M;
    double alpha;
}

```

```
{  
    for (int i = 0; i < N; i++) {  
        for (int j = 0; j < M; j++) {  
            Result [i][j] = alpha*A[i][j];  
        }  
    }  
}
```

---

## 2. METHODS OF MATRIX MULTIPLICATION

2.1. **The naive method.** The naive method of matrix multiplication is given by the definition itself:

$$(AB)_{ij} = \sum_{k=1}^P A_{ik} B_{kj}.$$

2.1.1. *NaivStandard.* First we implement this straightforward but we work with an auxiliary variable `aux` for the sum.

---

```
void NaivStandard(A, B, Result, N, P, M)
    double** A;
    double** B;
    double** Result;
    int N;
    int P;
    int M;
{
    double aux;

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            aux = 0.0;
            for (int k = 0; k < P; k++) {
                aux += A[i][k]*B[k][j];
            }
            Result[i][j] = aux;
        }
    }
}
```

---

2.1.2. *NaivOnArray.* A common error is to work all the time on the arrays itself instead of using an auxiliary variable. We only present this method to include it in the tests.

---

```
void NaivOnArray(A, B, Result, N, P, M)
    double** A;
    double** B;
    double** Result;
    int N;
    int P;
    int M;
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            Result[i][j] = 0.0;
            for (int k = 0; k < P; k++) {
                Result[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

---

2.1.3. *NaivKahan*. To improve the numerical stability of the process, we use the KAHAN trick (see [3]) for the summation. Instead of computing a sum like

$$\sum_{i=1}^n x_i$$

with

```
double sum = 0.0;
for( int i = 1; i <= n; i++){
    sum += x[i];
}
```

we use

```
double t;
double sum = 0.0;
double err = 0.0;
for( int i = 1; i <= n; i++){
    err = err + x[i];
    t = sum + err;
    err = (sum - t) + err;
    sum = t;
}
```

---

```
void NaivKahan(A, B, Result, N, P, M)
```

```
    double** A;
    double** B;
    double** Result;
    int N;
    int P;
    int M;
{
    double t, sum, err;

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            sum = 0.0;
            err = 0.0;
            for (int k = 0; k < P; k++) {
                err = err + A[i][k]*B[k][j];
                t = sum + err;
                err = (sum - t) + err;
                sum = t;
            }
            Result[i][j] = sum;
        }
    }
}
```

---

2.2. **Loop unrolling.** The method of *loop unrolling* doesn't reduce the abstract complexity of the matrix multiplication problem. It tries to reduce the overhead from the `for` loops and speed up the real computational work.

2.2.1. *NaivLoopUnrollingTwo*. Instead of computing

$$(AB)_{ij} = \sum_{k=1}^P A_{ik} B_{kj},$$

we use

$$(AB)_{ij} = \sum_{k=1}^{P/2} A_{i,2k-1}B_{2k-1,j} + A_{i,2k}B_{2k,j}, \quad \text{or}$$

$$(AB)_{ij} = \left( \sum_{k=1}^{\lfloor P/2 \rfloor} A_{i,2k-1}B_{2k-1,j} + A_{i,2k}B_{2k,j} \right) + A_{i,P}B_{P,j},$$

if  $P$  is even or odd, resp.

---

```

void NaivLoopUnrollingTwo(A, B, Result, N, P, M)
double** A;
double** B;
double** Result;
int N;
int P;
int M;
{

    int i, j, k;
    double aux;

    if (P % 2 == 0) {

        for (i = 0; i < N; i++) {
            for (j = 0; j < M; j++) {
                aux = 0.0;
                for (k = 0; k < P; k += 2) {
                    aux += A[i][k]*B[k][j] + A[i][k+1]*B[k+1][j];
                }
                Result[i][j] = aux;
            }
        }

    } else {

        int PP = P - 1;
        for (i = 0; i < N; i++) {
            for (j = 0; j < M; j++) {
                aux = 0.0;
                for (k = 0; k < PP; k += 2) {
                    aux += A[i][k]*B[k][j] + A[i][k+1]*B[k+1][j];
                }
                Result[i][j] = aux + A[i][PP]*B[PP][j];
            }
        }

    }

}

```

---

2.2.2. *NaivLoopUnrollingThree*. Like above we compute

$$(AB)_{ij} = \sum_{k=1}^{P/3} A_{i,3k-2}B_{3k-2,j} + A_{i,3k-1}B_{3k-1,j} + A_{i,3k}B_{3k,j}$$

instead of

$$(AB)_{ij} = \sum_{k=1}^P A_{ik} B_{kj},$$

if  $P$  is divisible by 3 (otherwise we use correction terms like above).

---

```

void NaivLoopUnrollingThree(A, B, Result, N, P, M)
double** A;
double** B;
double** Result;
int N;
int P;
int M;
{
    int i, j, k;
    double aux;

    if (P % 3 == 0) {
        for (i = 0; i < N; i++) {
            for (j = 0; j < M; j++) {
                aux = 0.0;
                for (k = 0; k < P; k += 3) {
                    aux += A[i][k]*B[k][j] + A[i][k+1]*B[k+1][j] + A[i][k+2]*B[k+2][j];
                }
                Result[i][j] = aux;
            }
        }

    } else if (P % 3 == 1) {
        int PP = P - 1;
        for (i = 0; i < N; i++) {
            for (j = 0; j < M; j++) {
                aux = 0.0;
                for (k = 0; k < PP; k += 3) {
                    aux += A[i][k]*B[k][j] + A[i][k+1]*B[k+1][j] + A[i][k+2]*B[k+2][j];
                }
                Result[i][j] = aux + A[i][PP]*B[PP][j];
            }
        }

    } else {
        int PP = P - 2;
        int PPP = P - 1;
        for (i = 0; i < N; i++) {
            for (j = 0; j < M; j++) {
                aux = 0.0;
                for (k = 0; k < PP; k += 3) {
                    aux += A[i][k]*B[k][j] + A[i][k+1]*B[k+1][j] + A[i][k+2]*B[k+2][j];
                }
                Result[i][j] = aux + A[i][PP]*B[PP][j] + A[i][PPP]*B[PPP][j];
            }
        }
    }
}

```



---

}

### 2.2.3. *NaivLoopUnrollingFour.*

---

```

void NaivLoopUnrollingFour(A, B, Result, N, P, M)
double** A;
double** B;
double** Result;
int N;
int P;
int M;
{

    int i, j, k;
    double aux;

    if (P % 4 == 0) {

        for (i = 0; i < N; i++) {
            for (j = 0; j < M; j++) {
                aux = 0.0;
                for (k = 0; k < P; k += 4) {
                    aux += A[i][k]*B[k][j] + A[i][k+1]*B[k+1][j] + A[i][k+2]*B[k+2][j]
                        + A[i][k+3]*B[k+3][j];
                }
                Result[i][j] = aux;
            }
        }

    } else if (P % 4 == 1) {

        int PP = P - 1;
        for (i = 0; i < N; i++) {
            for (j = 0; j < M; j++) {
                aux = 0.0;
                for (k = 0; k < PP; k += 4) {
                    aux += A[i][k]*B[k][j] + A[i][k+1]*B[k+1][j] + A[i][k+2]*B[k+2][j]
                        + A[i][k+3]*B[k+3][j];
                }
                Result[i][j] = aux + A[i][PP]*B[PP][j];
            }
        }

    } else if (P % 4 == 2) {

        int PP = P - 2;
        int PPP = P - 1;
        for (i = 0; i < N; i++) {
            for (j = 0; j < M; j++) {
                aux = 0.0;
                for (k = 0; k < PP; k += 4) {
                    aux += A[i][k]*B[k][j] + A[i][k+1]*B[k+1][j] + A[i][k+2]*B[k+2][j]
                        + A[i][k+3]*B[k+3][j];
                }
                Result[i][j] = aux + A[i][PP]*B[PP][j] + A[i][PPP]*B[PPP][j];
            }
        }

    } else {

```

```

int PP = P - 3;
int PPP = P - 2;
int PPPP = P - 1;
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        aux = 0.0;
        for (k = 0; k < PP; k += 4) {
            aux += A[i][k]*B[k][j] + A[i][k+1]*B[k+1][j] + A[i][k+2]*B[k+2][j]
                + A[i][k+3]*B[k+3][j];
        }
        Result[i][j] = aux + A[i][PP]*B[PP][j] + A[i][PPP]*B[PPP][j]
            + A[i][PPPP]*B[PPPP][j];
    }
}
}

```

---

### 2.3. Winograd's algorithm.

2.3.1. *WinogradOriginal*. From SHMUEL WINOGRAD we know a method that precomputes some values and reuse it in the computation of the entries of the result matrix. It is based on the fact that

$$(AB)_{ik} = \sum_{j=1}^{P/2} (A_{i,2j-1} + B_{2j,k})(A_{i,2j} + B_{2j-1,k}) - \underbrace{\sum_{j=1}^{P/2} A_{i,2j-1} A_{i,2j}}_{=:y_i} - \underbrace{\sum_{j=1}^{P/2} B_{2j-1,k} B_{2j,k}}_{=:z_k}$$

if  $P$  is even. In the case that  $P$  is odd, we use  $\lfloor P/2 \rfloor$  instead of  $P/2$  and add the missing product  $A_{i,P} B_{B,k}$  to each entry of the result matrix.

In the implementation we use `upsilon` to indicate if  $P$  is even and `gamma` as  $\lfloor P/2 \rfloor$ .

---

```

void WinogradOriginal(A, B, Result, N, P, M)
double** A;
double** B;
double** Result;
int N;
int P;
int M;
{
    int i, j, k;
    double aux;

    int upsilon = P % 2;
    int gamma = P - upsilon;

    double* y = malloc(M*sizeof(double));
    double* z = malloc(N*sizeof(double));

    for (i = 0; i < M; i++) {
        aux = 0.0;
        for (j = 0; j < gamma; j += 2) {
            aux += A[i][j]*A[i][j+1];
        }
        y[i] = aux;
    }
}

```

```

for (i = 0; i < N; i++) {
    aux = 0.0;
    for (j = 0; j < gamma; j += 2) {
        aux += B[j][i]*B[j+1][i];
    }
    z[i] = aux;
}

if (upsilon == 1) {

    /*
     * P is odd
     * The value A[i][P]*B[P][k] is missing in all auxiliary sums.
     */
    int PP = P-1;
    for (i = 0; i < M; i++) {
        for (k = 0; k < N; k++) {
            aux = 0.0;
            for (j = 0; j < gamma; j += 2) {
                aux += ( A[i][j] + B[j+1][k] )*( A[i][j+1] + B[j][k] );
            }
            Result[i][k] = aux - y[i] - z[k] + A[i][PP]*B[PP][k];
        }
    }

} else {

    /*
     * P is even
     * The result can be computed with the auxiliary sums.
     */
    for (i = 0; i < M; i++) {
        for (k = 0; k < N; k++) {
            aux = 0.0;
            for (j = 0; j < gamma; j += 2) {
                aux += ( A[i][j] + B[j+1][k] )*( A[i][j+1] + B[j][k] );
            }
            Result[i][k] = aux - y[i] - z[k];
        }
    }
}

free(y);
free(z);
}

```

---

2.3.2. *WinogradScaled*. From R. P. BRENT we know (see [1]) that the error of WINOGRAD's algorithm is

$$\|E\| \leq 2^{-\tau} \frac{N^2 + 12N - 8}{4} (\|A\| + \|B\|)^2,$$

where  $\tau$  is a parameter of the underlying number model (defined by WILKINSON in [5]) and  $N$  is size of the matrices (in the case that  $N = P = M$ ). Now suppose that  $\|A\|/\|B\| = k$ , then  $(\|A\| + \|B\|)^2 = (k + 2 + 1/k)\|A\|\|B\|$ , and so

$$\|E\| \leq 2^{-\tau} \frac{N^2 + 12N - 8}{4} (k + 2 + 1/k) \|A\| \|B\|.$$

According to BRENT it is always possible to find an integer  $\lambda$  such that

$$1/2 \leq \frac{2^\lambda \|A\|}{2^{-\lambda} \|B\|} \leq 2.$$

And because of

$$\max_{\frac{1}{2} \leq k \leq 2} k + 2 + 1/k = 9/2$$

he gets the bound

$$\|E\| \leq 2^{-\tau} \cdot \frac{9}{8} \cdot (N^2 + 12N - 8) \cdot \|A\| \cdot \|B\|$$

when multiplying the matrices  $2^\lambda A$  and  $2^{-\lambda} B$  (which doesn't change the result:  $(2^\lambda A) \cdot (2^{-\lambda} B) = 2^\lambda 2^{-\lambda} AB = AB$ ).

---

```

void WinogradScaled(A, B, Result, N, P, M)
    double** A;
    double** B;
    double** Result;
    int N;
    int P;
    int M;
{
    int i;

    /* Create scaled copies of A and B */
    double** CopyA = malloc(N*sizeof(double*));
    for( i = 0; i < N; i++){
        CopyA[i]=malloc(P*sizeof(double));
    }
    double** CopyB = malloc(P*sizeof(double*));
    for( i = 0; i < P; i++){
        CopyB[i]=malloc(M*sizeof(double));
    }

    /* Scaling factors */
    double a = NormInf(A, N, P);
    double b = NormInf(B, P, M);

    double lambda = floor(0.5 + log(b/a)/log(4));

    /* Scaling */
    MultiplyWithScalar(A, CopyA, N, P, pow(2, lambda));
    MultiplyWithScalar(B, CopyB, P, M, pow(2, -lambda));

    /* Using Winograd with scaled matrices */
    WinogradOriginal(CopyA, CopyB, Result, N, P, M);
}

```

---

**2.4. Strassen's algorithm.** The matrix multiplication algorithm from VOLKER STRASSEN is very famous. We present two variants.

**2.4.1. StrassenNaiv.** In his paper *Gaussian Elimination is not Optimal* (see [4]) STRASSEN developed a recursive algorithm  $\alpha_{m,k}$  for matrix multiplication for square matrices of order  $m2^k$ , where  $k, m \in \mathbb{N}$ . Let  $\alpha_{m,0}$  be the naive algorithm for matrix multiplication. We assume that  $\alpha_{m,k}$  is known, then we define  $\alpha_{m,k+1}$  as follows:

If  $A, B$  are matrices of order  $m2^{k+1}$  to be multiplied, we write

$$(1) \quad A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad \text{and} \quad C := AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

where  $A_{ij}, B_{ij}$  and  $C_{ij}$  are matrices of order  $m2^k$ . With the following auxiliary matrices

$$\begin{aligned} H_1 &:= (A_{11} + A_{22})(B_{11} + B_{22}) & H_2 &:= (A_{21} + A_{22})B_{11} \\ H_3 &:= A_{11}(B_{12} - B_{22}) & H_4 &:= A_{22}(B_{21} - B_{11}) \\ H_5 &:= (A_{11} + A_{12})B_{22} & H_6 &:= (A_{21} - A_{11})(B_{11} + B_{12}) \\ H_7 &:= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

of order  $m2^k$  computed using  $\alpha_{m,k}$  for matrix multiplication and the usual algorithm for addition and subtraction of matrices we get

$$\begin{aligned} C_{11} &= H_1 + H_4 - H_5 + H_7 & C_{12} &= H_3 + H_5 \\ C_{21} &= H_2 + H_4 & C_{22} &= H_1 + H_3 - H_2 + H_6. \end{aligned}$$

as the entries of the result matrix  $C$ .

To use the algorithm above we define  $X := \max\{N, P, M\}$  and set  $k := \lfloor \log_2 X \rfloor - 4$  and  $m := \lfloor X2^{-k} \rfloor + 1$ . Now with  $Y := m2^k$  we define matrices  $\tilde{A}$  and  $\tilde{B}$  of size  $Y \times Y$ . The main idea is, to embed the given matrices into  $Y \times Y$  matrices, by adding zero rows and columns. Now we use the STRASSEN algorithm to compute  $\tilde{A}\tilde{B}$  and extract the result  $AB$  from it, by removing the additional rows and columns.

In the implementation we use ResultPart= $C$ , Aux1= $H_1$ , ..., Aux7= $H_7$ , AuxResult= $\tilde{A}\tilde{B}$ , MaxSize= $X$  and NewSize= $Y$ .

---

```

void StrassenNaiv(A, B, Result, N, P, M)
    double** A;
    double** B;
    double** Result;
    int N;
    int P;
    int M;
{

    int MaxSize, k, m, NewSize, i, j;

    MaxSize = max(N,P);
    MaxSize = max(MaxSize,M);

    if ( MaxSize < 16){
        MaxSize = 16; // otherwise it is not possible to compute k
    }

    k = floor(log(MaxSize)/log(2)) - 4;
    m = floor(MaxSize * pow(2,-k)) + 1;

    NewSize = m * pow(2,k);

    // add zero rows and columns to use Strassens algorithm

    double** NewA = malloc(NewSize*sizeof(double));
    double** NewB = malloc(NewSize*sizeof(double));
    double** AuxResult = malloc(NewSize*sizeof(double));
    for( i = 0; i < NewSize; i++){
        NewA[i] = malloc(NewSize*sizeof(double));
        NewB[i] = malloc(NewSize*sizeof(double));
        AuxResult[i] = malloc(NewSize*sizeof(double));
    }
}

```

```

    for( i = 0; i < NewSize; i++){
        for( j = 0; j < NewSize; j++){
            NewA[i][j] = 0.0;
            NewB[i][j] = 0.0;
        }
    }

    for( i = 0; i < N; i++){
        for( j = 0; j < P; j++){
            NewA[i][j] = A[i][j];
        }
    }

    for( i = 0; i < P; i++){
        for( j = 0; j < M; j++){
            NewB[i][j] = B[i][j];
        }
    }

    StrassenNaivStep(NewA, NewB, AuxResult, NewSize, m);

    // extract the result

    for( i = 0; i < N; i++){
        for( j = 0; j < M; j++){
            Result[i][j] = AuxResult[i][j];
        }
    }
}

```

---

```

void StrassenNaivStep(A, B, Result, N, m)
    double** A;
    double** B;
    double** Result;
    int N;
    int m; //to reduce the recursion; idea from Strassen
{

    int i, j, NewSize;

    if( (N % 2 == 0) && (N > m) ){ // recursive use of StrassenNaivStep

        NewSize = N / 2;

        // decompose A and B
        // create ResultPart, Aux1,...,Aux7 and Helper1, Helper2
        double** A11 = malloc(NewSize*sizeof(double*));
        double** A12 = malloc(NewSize*sizeof(double*));
        double** A21 = malloc(NewSize*sizeof(double*));
        double** A22 = malloc(NewSize*sizeof(double*));
        double** B11 = malloc(NewSize*sizeof(double*));
        double** B12 = malloc(NewSize*sizeof(double*));
        double** B21 = malloc(NewSize*sizeof(double*));
        double** B22 = malloc(NewSize*sizeof(double*));
        double** ResultPart11 = malloc(NewSize*sizeof(double*));
        double** ResultPart12 = malloc(NewSize*sizeof(double*));

```

```

double** ResultPart21 = malloc(NewSize*sizeof(double*));
double** ResultPart22 = malloc(NewSize*sizeof(double*));
double** Helper1 = malloc(NewSize*sizeof(double*));
double** Helper2 = malloc(NewSize*sizeof(double*));
double** Aux1 = malloc(NewSize*sizeof(double*));
double** Aux2 = malloc(NewSize*sizeof(double*));
double** Aux3 = malloc(NewSize*sizeof(double*));
double** Aux4 = malloc(NewSize*sizeof(double*));
double** Aux5 = malloc(NewSize*sizeof(double*));
double** Aux6 = malloc(NewSize*sizeof(double*));
double** Aux7 = malloc(NewSize*sizeof(double*));
for( i = 0; i < NewSize; i++){
    A11[i] = malloc(NewSize*sizeof(double));
    A12[i] = malloc(NewSize*sizeof(double));
    A21[i] = malloc(NewSize*sizeof(double));
    A22[i] = malloc(NewSize*sizeof(double));
    B11[i] = malloc(NewSize*sizeof(double));
    B12[i] = malloc(NewSize*sizeof(double));
    B21[i] = malloc(NewSize*sizeof(double));
    B22[i] = malloc(NewSize*sizeof(double));
    ResultPart11[i] = malloc(NewSize*sizeof(double));
    ResultPart12[i] = malloc(NewSize*sizeof(double));
    ResultPart21[i] = malloc(NewSize*sizeof(double));
    ResultPart22[i] = malloc(NewSize*sizeof(double));
    Helper1[i] = malloc(NewSize*sizeof(double));
    Helper2[i] = malloc(NewSize*sizeof(double));
    Aux1[i] = malloc(NewSize*sizeof(double));
    Aux2[i] = malloc(NewSize*sizeof(double));
    Aux3[i] = malloc(NewSize*sizeof(double));
    Aux4[i] = malloc(NewSize*sizeof(double));
    Aux5[i] = malloc(NewSize*sizeof(double));
    Aux6[i] = malloc(NewSize*sizeof(double));
    Aux7[i] = malloc(NewSize*sizeof(double));
}

// fill new matrices

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        A11[i][j] = A[i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        A12[i][j] = A[i][NewSize + j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        A21[i][j] = A[NewSize + i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        A22[i][j] = A[NewSize + i][NewSize + j];
    }
}

```

```

    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        B11[i][j] = B[i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        B12[i][j] = B[i][NewSize + j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        B21[i][j] = B[NewSize + i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        B22[i][j] = B[NewSize + i][NewSize + j];
    }
}

// computing the seven aux. variables

Plus(A11, A22, Helper1, NewSize, NewSize);
Plus(B11, B22, Helper2, NewSize, NewSize);
StrassenNaivStep(Helper1, Helper2, Aux1, NewSize, m);

Plus(A21, A22, Helper1, NewSize, NewSize);
StrassenNaivStep(Helper1, B11, Aux2, NewSize, m);

Minus(B12, B22, Helper1, NewSize, NewSize);
StrassenNaivStep(A11, Helper1, Aux3, NewSize, m);

Minus(B21, B11, Helper1, NewSize, NewSize);
StrassenNaivStep(A22, Helper1, Aux4, NewSize, m);

Plus(A11, A12, Helper1, NewSize, NewSize);
StrassenNaivStep(Helper1, B22, Aux5, NewSize, m);

Minus(A21, A11, Helper1, NewSize, NewSize);
Plus(B11, B12, Helper2, NewSize, NewSize);
StrassenNaivStep(Helper1, Helper2, Aux6, NewSize, m);

Minus(A12, A22, Helper1, NewSize, NewSize);
Plus(B21, B22, Helper2, NewSize, NewSize);
StrassenNaivStep(Helper1, Helper2, Aux7, NewSize, m);

// computing the four parts of the result

Plus(Aux1, Aux4, ResultPart11, NewSize, NewSize);
Minus(ResultPart11, Aux5, ResultPart11, NewSize, NewSize);
Plus(ResultPart11, Aux7, ResultPart11, NewSize, NewSize);

```



```

Plus(Aux3, Aux5, ResultPart12, NewSize, NewSize);

Plus(Aux2, Aux4, ResultPart21, NewSize, NewSize);

Plus(Aux1, Aux3, ResultPart22, NewSize, NewSize);
Minus(ResultPart22, Aux2, ResultPart22, NewSize, NewSize);
Plus(ResultPart22, Aux6, ResultPart22, NewSize, NewSize);

// store results in the "result matrix"

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        Result[i][j] = ResultPart11[i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        Result[i][NewSize + j] = ResultPart12[i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        Result[NewSize + i][j] = ResultPart21[i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        Result[NewSize + i][NewSize + j] = ResultPart22[i][j];
    }
}

// free helper variables

free(A11);
free(A12);
free(A21);
free(A22);

free(B11);
free(B12);
free(B21);
free(B22);

free(ResultPart11);
free(ResultPart12);
free(ResultPart21);
free(ResultPart22);

free(Helper1);
free(Helper2);

free(Aux1);
free(Aux2);
free(Aux3);

```

```

    free(Aux4);
    free(Aux5);
    free(Aux6);
    free(Aux7);

} else {

    // use naiv algorithm
    NaivStandard(A, B, Result, N, N, N);

}

```

---

2.4.2. *StrassenWinograd*. In the paper *Efficient Procedures for using Matrix Algorithms* (see [2]) PATRICK C. FISCHER and ROBERT L. PROBERT discussed an idea of SHMUEL WINOGRAD which reduces the number of used additions to 15 (instead of 18 in *StrassenNaiv*).

Like above, we define a recursive algorithm  $\alpha$  for multiplication of square matrices of order  $m2^{k+1}$ . Let  $A$  and  $B$  be matrices of this size. We assume that  $\alpha$  is already known for the order  $m2^k$ . We define  $C := AB$  and decompose  $A$ ,  $B$  and  $C$  according to Equation (1). We define

$$\begin{aligned} A_1 &:= A_{11} - A_{21} & B_1 &:= B_{22} - B_{12} \\ A_2 &:= A_{22} - A_1 & B_2 &:= B_1 + B_{11} \end{aligned}$$

and

$$\begin{aligned} H_1 &= A_{11}B_{11} & H_5 &= A_1B_1 \\ H_2 &= A_{12}B_{21} & H_6 &= (A_{12} - A_2)B_{22} \\ H_3 &= A_2B_2 & H_7 &= A_{22}(B_{21} - B_2) \\ H_4 &= (A_{21} + A_{22})(B_{12} - B_{11}). \end{aligned}$$

With

$$H_8 := H_1 + H_3 \qquad H_9 := H_8 + H_4$$

we finally get

$$\begin{aligned} C_{11} &= H_1 + H_2 & C_{12} &= H_9 + H_6 \\ C_{21} &= H_8 + H_5 + H_7 & C_{22} &= H_9 + H_5. \end{aligned}$$


---

```

void StrassenWinograd(A, B, Result, N, P, M)
double** A;
double** B;
double** Result;
int N;
int P;
int M;
{
    int MaxSize, k, m, NewSize, i, j;

    MaxSize = max(N,P);
    MaxSize = max(MaxSize,M);

    if ( MaxSize < 16){
        MaxSize = 16; // otherwise it is not possible to compute k
    }

    k = floor(log(MaxSize)/log(2)) - 4;

```

```

m = floor(MaxSize * pow(2,-k)) + 1;

NewSize = m * pow(2,k);

// add zero rows and columns to use Strassens algorithm

double** NewA = malloc(NewSize*sizeof(double*));
double** NewB = malloc(NewSize*sizeof(double*));
double** AuxResult = malloc(NewSize*sizeof(double*));
for( i = 0; i < NewSize; i++){
    NewA[i] = malloc(NewSize*sizeof(double));
    NewB[i] = malloc(NewSize*sizeof(double));
    AuxResult[i] = malloc(NewSize*sizeof(double));
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        NewA[i][j] = 0.0;
        NewB[i][j] = 0.0;
    }
}

for( i = 0; i < N; i++){
    for( j = 0; j < P; j++){
        NewA[i][j] = A[i][j];
    }
}

for( i = 0; i < P; i++){
    for( j = 0; j < M; j++){
        NewB[i][j] = B[i][j];
    }
}

StrassenWinogradStep(NewA, NewB, AuxResult, NewSize, m);

// extract the result

for( i = 0; i < N; i++){
    for( j = 0; j < M; j++){
        Result[i][j] = AuxResult[i][j];
    }
}
}



---


void StrassenWinogradStep(A, B, Result, N, m)
double** A;
double** B;
double** Result;
int N;
int m; //to reduce the recursion; idea from Strassen
{

    int i, j, NewSize;

    if( (N % 2 == 0) && (N > m) ){ // recursive use of StrassenNaivStep

```

```

NewSize = N / 2;

// decompose A and B
// create ResultPart, Aux1,...,Aux7 and Helper1, Helper2
double** A11 = malloc(NewSize*sizeof(double));
double** A12 = malloc(NewSize*sizeof(double));
double** A21 = malloc(NewSize*sizeof(double));
double** A22 = malloc(NewSize*sizeof(double));
double** B11 = malloc(NewSize*sizeof(double));
double** B12 = malloc(NewSize*sizeof(double));
double** B21 = malloc(NewSize*sizeof(double));
double** B22 = malloc(NewSize*sizeof(double));
double** A1 = malloc(NewSize*sizeof(double));
double** A2 = malloc(NewSize*sizeof(double));
double** B1 = malloc(NewSize*sizeof(double));
double** B2 = malloc(NewSize*sizeof(double));
double** ResultPart11 = malloc(NewSize*sizeof(double));
double** ResultPart12 = malloc(NewSize*sizeof(double));
double** ResultPart21 = malloc(NewSize*sizeof(double));
double** ResultPart22 = malloc(NewSize*sizeof(double));
double** Helper1 = malloc(NewSize*sizeof(double));
double** Helper2 = malloc(NewSize*sizeof(double));
double** Aux1 = malloc(NewSize*sizeof(double));
double** Aux2 = malloc(NewSize*sizeof(double));
double** Aux3 = malloc(NewSize*sizeof(double));
double** Aux4 = malloc(NewSize*sizeof(double));
double** Aux5 = malloc(NewSize*sizeof(double));
double** Aux6 = malloc(NewSize*sizeof(double));
double** Aux7 = malloc(NewSize*sizeof(double));
double** Aux8 = malloc(NewSize*sizeof(double));
double** Aux9 = malloc(NewSize*sizeof(double));
for( i = 0; i < NewSize; i++){
    A11[i] = malloc(NewSize*sizeof(double));
    A12[i] = malloc(NewSize*sizeof(double));
    A21[i] = malloc(NewSize*sizeof(double));
    A22[i] = malloc(NewSize*sizeof(double));
    B11[i] = malloc(NewSize*sizeof(double));
    B12[i] = malloc(NewSize*sizeof(double));
    B21[i] = malloc(NewSize*sizeof(double));
    B22[i] = malloc(NewSize*sizeof(double));
    A1[i] = malloc(NewSize*sizeof(double));
    A2[i] = malloc(NewSize*sizeof(double));
    B1[i] = malloc(NewSize*sizeof(double));
    B2[i] = malloc(NewSize*sizeof(double));
    ResultPart11[i] = malloc(NewSize*sizeof(double));
    ResultPart12[i] = malloc(NewSize*sizeof(double));
    ResultPart21[i] = malloc(NewSize*sizeof(double));
    ResultPart22[i] = malloc(NewSize*sizeof(double));
    Helper1[i] = malloc(NewSize*sizeof(double));
    Helper2[i] = malloc(NewSize*sizeof(double));
    Aux1[i] = malloc(NewSize*sizeof(double));
    Aux2[i] = malloc(NewSize*sizeof(double));
    Aux3[i] = malloc(NewSize*sizeof(double));
    Aux4[i] = malloc(NewSize*sizeof(double));
    Aux5[i] = malloc(NewSize*sizeof(double));
    Aux6[i] = malloc(NewSize*sizeof(double));
    Aux7[i] = malloc(NewSize*sizeof(double));
    Aux8[i] = malloc(NewSize*sizeof(double));

```

```

    Aux9[i] = malloc(NewSize*sizeof(double));
}

// fill new matrices

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        A11[i][j] = A[i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        A12[i][j] = A[i][NewSize + j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        A21[i][j] = A[NewSize + i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        A22[i][j] = A[NewSize + i][NewSize + j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        B11[i][j] = B[i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        B12[i][j] = B[i][NewSize + j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        B21[i][j] = B[NewSize + i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        B22[i][j] = B[NewSize + i][NewSize + j];
    }
}

// computing the 4 + 9 aux. variables

Minus(A11, A21, A1, NewSize, NewSize);
Minus(A22, A1, A2, NewSize, NewSize);
Minus(B22, B12, B1, NewSize, NewSize);

```

```

Plus(B1, B11, B2, NewSize, NewSize);

StrassenWinogradStep(A11, B11, Aux1, NewSize, m);
StrassenWinogradStep(A12, B21, Aux2, NewSize, m);
StrassenWinogradStep(A2, B2, Aux3, NewSize, m);
Plus(A21, A22, Helper1, NewSize, NewSize);
Minus(B12, B11, Helper2, NewSize, NewSize);
StrassenWinogradStep(Helper1, Helper2, Aux4, NewSize, m);
StrassenWinogradStep(A1, B1, Aux5, NewSize, m);
Minus(A12, A2, Helper1, NewSize, NewSize);
StrassenWinogradStep(Helper1, B22, Aux6, NewSize, m);
Minus(B21, B2, Helper1, NewSize, NewSize);
StrassenWinogradStep(A22, Helper1, Aux7, NewSize, m);
Plus(Aux1, Aux3, Aux8, NewSize, NewSize);
Plus(Aux8, Aux4, Aux9, NewSize, NewSize);

// computing the four parts of the result

Plus(Aux1, Aux2, ResultPart11, NewSize, NewSize);

Plus(Aux9, Aux6, ResultPart12, NewSize, NewSize);

Plus(Aux8, Aux5, Helper1, NewSize, NewSize);
Plus(Helper1, Aux7, ResultPart21, NewSize, NewSize);

Plus(Aux9, Aux5, ResultPart22, NewSize, NewSize);

// store results in the "result matrix"

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        Result[i][j] = ResultPart11[i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        Result[i][NewSize + j] = ResultPart12[i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        Result[NewSize + i][j] = ResultPart21[i][j];
    }
}

for( i = 0; i < NewSize; i++){
    for( j = 0; j < NewSize; j++){
        Result[NewSize + i][NewSize + j] = ResultPart22[i][j];
    }
}

// free helper variables

free(A11);
free(A12);
free(A21);

```

```
    free(A22);

    free(B11);
    free(B12);
    free(B21);
    free(B22);

    free(A1);
    free(A2);
    free(B1);
    free(B2);

    free(ResultPart11);
    free(ResultPart12);
    free(ResultPart21);
    free(ResultPart22);

    free(Helper1);
    free(Helper2);

    free(Aux1);
    free(Aux2);
    free(Aux3);
    free(Aux4);
    free(Aux5);
    free(Aux6);
    free(Aux7);
    free(Aux8);
    free(Aux9);

} else {

    // use naiv algorithm
    NaivStandard(A, B, Result, N, N, N);

}
```

---

## 3. TESTS

To test the algorithms we compute the product of a random square matrix  $A$  with its scaled variant  $B := 8A$ . The test routine is part of the package of C files. It has the command line options

- -O matrix size, *standard*: 400
- -R number of repeats, *standard*: 10

TIME TEST FOR METHODS OF MATRIX MULTIPLICATION		
C = A*(8A), where A is a (n x n) random matrix with n = 200		
method	time (sec)	NormInf( N-C )
N := NaivKahan	0.146027	
NaivStandard	0.083263	0.0000000000
NaivOnArray	0.121735	0.0000000000
NaivLoopUnrollingTwo	0.072981	0.0000000000
NaivLoopUnrollingThree	0.068356	0.0000000000
NaivLoopUnrollingFour	0.067080	0.0000000000
StrassenNaiv	0.077584	0.0000000001
StrassenWinograd	0.080180	0.0000000000
WinogradOriginal	0.073193	0.0000000001
WinogradScaled	0.061350	0.0000000001

TIME TEST FOR METHODS OF MATRIX MULTIPLICATION		
C = A*(8A), where A is a (n x n) random matrix with n = 800		
method	time (sec)	NormInf( N-C )
N := NaivKahan	11.817480	
NaivStandard	7.370365	0.0000000009
NaivOnArray	10.828352	0.0000000009
NaivLoopUnrollingTwo	6.914036	0.0000000006
NaivLoopUnrollingThree	6.605404	0.0000000005
NaivLoopUnrollingFour	6.480659	0.0000000004
StrassenNaiv	3.806864	0.0000000022
StrassenWinograd	3.978543	0.0000000010
WinogradOriginal	6.892913	0.0000000036
WinogradScaled	5.781587	0.0000000022



## REFERENCES

- [1] BRENT, R. P.: *Algorithms for Matrix Multiplication*. STAN-CS-70-157. Computer Science Department, School of Humanities and Sciences, Stanford University. March 1970.
- [2] FISCHER, PATRICK C. and PROBERT, ROBERT L.: *Efficient Procedures for using Matrix Algorithms*.
- [3] KAHAN, WILLIAM: *Further remarks on reducing truncation errors*. Communications of the ACM 8 (1): 40. January 1965.
- [4] STRASSEN, VOLKER: *Gaussian Elimination is not Optimal*. Numer. Math. 13: 354–356. 1969.
- [5] WILKINSON, J. H.: *Rounding Errors in Algebraic Processes*. H.M.S.O. 1963.

IVO HEDTKE, RESEARCH GROUP: DATA STRUCTURES AND EFFICIENT ALGORITHMS, INSTITUTE FOR COMPUTER SCIENCE, MARTIN-LUTHER-UNIVERSITÄT HALLE-WITTENBERG, VON-SECKENDORFF-PLATZ 1, 06120 HALLE, GERMANY  
*E-mail address:* Ivo.Hedtke@uni-jena.de